Evil Hangman Project

Program Description: Evil Hangman is a program that actively cheats at Hangman. Instead of choosing a single word that the player tries to guess, the program maintains a set of words that it continuously pares down. It does the latter in such a way as to minimize the player's chance of winning.

For this program we assume a word is a sequence of letters a-z. We also assume that letter comparisons are case-insensitive. Thus, 'a' matches either 'a' or 'A', and 'A' matches either 'a' or 'A'.

How the Program Works:

1. Run your program using the following usage statement:

   Usage: java [your main class name] *dictionary wordLength guesses*

   dictionary is the path to a text file with whitespace separated words (no numbers, punctuation, etc.)
   wordLength is an integer ≥ 2.
   guesses is an integer ≥ 1.

2. Begin with a set of English words. In theory, the set contains all the words in the English language, but your set will be initialized with contents from a dictionary file.

3. Run the game:
   a. At the beginning of each turn, display the number of remaining guesses, an alphabetized list of letters guessed so far, and the partially constructed word.
   b. Prompt the user for his/her next letter guess. If the guess is not an upper- or lower-case letter then print "Invalid input" on the next line and reprompt. If the user has already guessed the letter print "You already used that letter" on the next line and reprompt.
   c. Report the number of position(s) in which the letter appears. If the current word list doesn't contain any word with that letter, decrement the number of remaining guesses and print "`Sorry, there are no <inputLetter>'s`" on a new line (replace *inputLetter* with the actual letter input). If all words in the list contain at least one word with that letter, print "`Yes, there is <number> <letter>`" where *letter* is the letter guessed and *numbers* is the number of times *letter* appears in the word with the fewest number of those letters.
   d. If there are more turns left, repeat step 3 (this step). Remember, the number of turns only decrements when there is at least one word in the list that does not contain the guessed letter.

6. When the user runs out of guesses, pick a word from the word list and display it as the word that the computer initially "chose." If the user correctly guesses the word, before he/she runs out of guesses, display "You Win!" and display the correct word.

The "Evil" Algorithm:

1. Begin with the set of English words, B, read in from the indicated input file.
2. Assuming the user inputs a length *l*, Create a subset of words, L, such that every word in B of length *l* is in L and L ⊆ S.
3. Each time the user guesses a letter:
    a. Partition the word list into "word groups" based on the positions of the guessed letter in the words.

    For example, let the current word list be [ALLY, BEST, COOL, DEAL, ECHO, ELSE, FLEW, GOOD, HEAL, HOPE, LAZY]. Assume the player guesses the letter "E." Your program should partition the word list into the following six word families:

    1. *E---* *contains **E**CHO.*
    2. *-E--* *contains B**E**ST, D**E**AL, H**E**AL.*
    3. *--E-* *contains FL**E**W.*
    4. *---E* *contains HOP**E**.*
    5. *E--E* *contains **E**LS**E**.*
    6. *----* *contains ALLY, COOL, GOOD, LAZY.*

    b. Choose the largest of these word groups to replace L.

    In the example above, the largest word group is the one of the form ----.

    If two or more of the groups are of the same size, choose the one to return according to the following priorities:

    1. Choose the group in which the letter does not appear at all.
    2. If each group has the guessed letter, choose the one with the fewest letters.
    3. If this still has not resolved the issue, choose the one with the rightmost guessed letter.
    4. If there is still more than one group, choose the one with the next rightmost letter. Repeat this step (step 4) until a group is chosen.

Deliverables:

Build a class that correctly implements the `IEvilHangmanGame` interface according to this specification. This class should have a constructor that takes no arguments. The passoff driver will call methods on this class to test your implementation. The `IEvilHangmanGame` interface is provided on the course web site in the files associated with this project.

Also create a class containing a main method for running your game from the command line. This main method should function as described previously. When the TA's pass you off, they will run your main method manually from the command line to make sure your game works.

Error Handling:

Errors that could occur include an empty dictionary file, bad command line parameters, and bad characters in the dictionary file. You may handle these errors in any way you like as long as the `IEvilHangmanGame` interface is implemented correctly. We will not test for any errors or unexpected input during the first part of the passoff (viz. while we are checking that your "evil" algorithm is working properly), including the `GuessAlreadyMadeException`. However, when we call your `main` method to play your version of the game, you will want to consider all of the possible inputs that a user could do and plan for them accordingly. (ie. Is the input a number? a special character [like `$` or `#` or `!`, etc…]? a capital letter? a newline? a space? a string of more than one characters? etc.…)

Hints & Things to Consider:

1. Consider which data structure would best handle the operations you need to perform on the word set.
2. Letter position matters just as much as letter frequency. For example, "DEER" and "HERE" are in two different families even though they both have two E's in them.
3. Don't explicitly enumerate word families. If you are working with a word of length n, then there are 2^n possible word families for each letter. However, most of these families don't actually appear in the English language. For example, no English words contain three consecutive U's, and no word matches the pattern E-EE-EE-E. There will be no words in the input file that are cannot be found in English dictionaries.
4. Be careful to consider which parts of your code you need to be stateless and which parts you want to maintain state from one method call to another. (viz. You need to update your current dictionary of possible words after every call of `makeGuess`)

Example Game:

```
java EvilHangman dictionary.txt 5 10
You have 10 guesses left
Used letters:
Word: -----
Enter guess: a
Sorry, there are no a's

You have 9 guesses left
Used letters: a
Word: -----
Enter guess: e
Sorry, there are no e's

You have 8 guesses left
Used letters: a e
Word: -----
Enter guess: i
Sorry, there are no i's

You have 7 guesses left
Used letters: a e i
Word: -----
Enter guess: o
Sorry, there are no o's

You have 6 guesses left
Used letters: a e i o
Word: -----
Enter guess: u
Yes, there is 1 u

You have 6 guesses left
Used letters: a e i o u
Word: -u---
Enter guess: f
Sorry, there are no f's

You have 5 guesses left
Used letters: a e f i o u
Word: -u---
Enter guess: t
Sorry, there are no t's
```

```
You have 4 guesses left
Used letters: a e f i o t u
Word: -u---
Enter guess: m
Sorry, there are no m's

You have 3 guesses left
Used letters: a e f i m o t u
Word: -u---
Enter guess: l
Sorry, there are no l's

You have 2 guesses left
Used letters: a e f i l m o t u
Word: -u---
Enter guess: s
Sorry, there are no s's

You have 1 guess left
Used letters: a e f i l m o s t u
Word: -u---
Enter guess: b
You lose!
The word was: vuggy
```